VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁSTROJ PRO AUTOMATIZOVANÉ TESTOVÁNÍ GUI

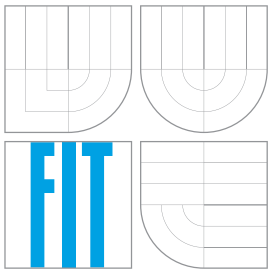
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ VACEK

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

NÁSTROJ PRO AUTOMATIZOVANÉ TESTOVÁNÍ GUI

A TOOL FOR AUTOMATED TESTING OF GUI

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ VACEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. ALEŠ SMRČKA, Ph.D.

BRNO 2014

Abstrakt

Testování GUI je navzdory jeho použitelnosti a rozšířenosti poměrně nově vznikající obor. Pro ověření GUI funkčnosti se používá často ručního testování. Cílem práce je vytvořit nástroj, který lze využít k otestování a ovládání GUI. Nástrojem bude knihovna pro automatizované testování GUI s využitím metody rozpoznávání objektů. Knihovna bude detekovat základní objekty GUI a manipulovat s nimi podle jejich typického chování. Detekce objektu bude založena na zpracování obrazu a pozorování grafických změn objektu při příchozích událostech od klávesnice a myši.

Abstract

Despite of GUI usability and availability is GUI testing quite new specialization technique. Manual testing is often used for verifying of GUI functionality. The aim of this work is to create a tool for testing and controling GUI. By tool is meant library for automated testing of GUI using object recognization method. Library detects basic GUI elements and controls them according to their standard behavior. Object detection depends on image processing and tracing graphic changes while mouse and key events are incoming.

Klíčová slova

automatizované testování, testování GUI, VNC, virtualizace, zpracování obrazu

Keywords

test automation, GUI testing, VNC, virtualization, image processing

Citace

Lukáš Vacek: Nástroj pro automatizované testování GUI, bakalářská práce, Brno, FIT VUT v Brně, 2014

Nástroj pro automatizované testování GUI

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky, Ph.D.

.....
Lukáš Vacek
21. května 2014

Poděkování

Děkuji mému vedoucímu Ing. Aleši Smrčkovi za jeho pomoc během psaní této práce. Dále děkuji technické podpoře Libvirt za rychlou a užitečnou komunikaci při řešení některých problémů.

Obsah

Kapitola 1

Úvod

Ještě v nedávné minulosti se na testování softwaru nekladly příliš velké požadavky. Pro většinu firem vyvíjející software nebylo nutné, aby většina jejich zaměstnanců byla seznámena s technickými principy a procesy testování. S postupem času, kdy se software stával nedílnou součástí každodenního života, vznikaly také přísnější požadavky na jeho spolehlivost, udržovatelnost a bezpečnost. Zvyšující se požadavky měly za následek změnu způsobu testování softwaru, která zahrnovala zvýšení technické znalosti ze strany softwarových inženýrů a větší důraz na testování od vývojářů.

V dnešní době je testování jednou z nejdůležitějších součástí životního cyklu softwaru. Chyby v produktu mohou vést k selhání celého systému a způsobit tak velké finanční nebo i lidské škody.

Kromě odhalování chyb slouží testování také k zjištění a ověření funkčnosti produktu, minimalizování rizika a předcházení problémům při tvorbě a provozu softwaru. Testování by se tedy nemělo brát na lehkou váhu, proto ve většině firem vyvíjející software existují specializované skupiny lidí, tzv. *Quality Assurance*, kteří se zabývají důkladným testováním.

Důležitým aspektem softwaru je jeho uživatelské rozhraní. Uživatelské rozhraní je určeno ke komunikaci a ovládání příslušné aplikace. Mezi nejpoužívanější patří textové a grafické rozhraní. Textové uživatelské rozhraní poskytuje uživateli výstup (výsledek) na základě jeho vstupu (požadavku). Grafické uživatelské rozhraní zajišťuje komunikaci mezi aplikací a uživatelem, který ji obsluhuje. Grafické uživatelské rozhraní je v porovnáním s textovým více intuitivní, ale také složitější pro testování, protože v jednom okamžiku se může vykonat více událostí. Tato bakalářská práce se bude zabývat právě testováním grafického uživatelského rozhraní(anglicky *Graphical User Interface*, dále označováno jako GUI). GUI se skládá z objektů, jako jsou například tlačítka, ikony, zaškrtávací políčka atd. Grafické objekty jsou hierarchicky uspořádány a na základě vzhledu by měl uživatel očekávat, jakou funkčnost nabízejí. Uživatel ovládáním zasílá objektům události, které jsou dále zpracovány. Při tomto zpracování může dojít ke změně stavu softwaru nebo ke grafické změně některých objektů.

V následující kapitole se seznámíme se základy testování. Zmíněno bude rozdělení testovacích principů a jejich výhody i nevýhody. Kapitola 3 popisuje požadavky nutné pro používání knihovny. Návrhu a implementaci knihovny pro automatizované testování GUI jsou věnovány kapitoly 4 a 5. Návrh obsahuje rozdělení knihovny na moduly, jejich rozhraní a popis komunikace modulů. V implementační části jsou popsány použité technologie a implementační problémy s jejich řešením. Zjištěné výsledky jsou pak prezentovány a shrnuty v kapitole číslo 6. Závěrečná kapitola obsahuje zhodnocení dosažených výsledků s návrhy na další možné rozšíření této knihovny. V oblasti testování se často používají anglické názvy,

proto budu k překládaným termínům zmiňovat i jejich anglický název.

Cílem této práce je vytvořit knihovnu v jazyce C++ pro automatizované testování GUI s využitím metody rozpoznávání objektů na obrazovce. Pomocí knihovny se spustí testovaná aplikace a proběhne rozpoznání jejího GUI. S rozpoznanými komponentami lze manipulovat podle jejich typického chování. Po provedené aktivitě může dojít ke změnám v GUI. Analýzou těchto změn lze zjistit, co se ve scéně stalo a jestli GUI objekt pracuje správně. Takovým postupem může uživatel otestovat některé testovací případy.

Kapitola 2

Stručný úvod do testování

Testování je dle [7] definováno jako proces vykonávání vybrané sady testů nad softwarovou komponentou se záměrem odhalení chyb a vyhodnocení kvality.

2.1 Analýza programu

Analýza počítačového programu je proces automatického analyzování chování programu za účelem optimalizace a odstranění nedostatků. Analýza je rozdělena na statickou a dynamickou.

2.1.1 Statická analýza

Statická analýza zkoumá artefakty kódu manuálně nebo pomocí nástrojů, aniž by byl program spuštěn [7]. Využívá se při překládání zdrojového kódu nebo při procházení a inspekci kódu.

2.1.2 Dynamická analýza

Při dynamické analýze je softwarový artefakt spuštěn s určitými vstupními hodnotami a chování s výstupy jsou porovnávány s tím, co se očekávalo. Dynamická analýza může být použita pouze na zdrojový kód. Používá se k odhalení chyb a zjišťování kvality kódu [10]. Dynamické analýzy se využívá při testování jednotek, integračním testování, systémovém testování apod.

2.2 Testovací přístupy

V kapitole jsou popsány základní testovací přístupy s jejich principy, výhodami a nevýhodami.

2.2.1 Metoda black-box

Na testovaný software se nahlíží jako na černou skříňku. Tester nemá představu o tom, jak je implementován a jaké obsahuje vnitřní datové a programové struktury. Při testování se využívá pouze rozhraní aplikace. Na základě porovnání aktuálního výstupu s očekávaným výstupem se určí, jestli test prošel nebo selhal. Správné chování aplikace je popsáno

ve specifikaci, kterou má tester k dispozici. Nevýhodou je, že i když je výstup očekávaný, neznamená to, že aplikace je napsána správně a efektivně.

2.2.2 Metoda white-box

Na rozdíl od metody black-box je při testování využíváno znalostí o vnitřních datových a programových strukturách. Dostupné jsou veškeré informace o testované aplikaci (zdrojový kód, dokumentace). Na základě analýzy zdrojového kódu se vytvoří testovací případy, které splňují požadované kritérium pokrytí. Metoda se nejčastěji používá pro jednotkové testování.

2.2.3 Metoda grey-box

Grey-box je kombinací metod white-box a black-box. Tester zná některé vnitřní struktury, ale není s nimi obeznámen hluboce do detailů. Výhodou metody je, že tester má představu o tom, co se uvnitř aplikace odehrává a může tak lépe otestovat produkt.

2.2.4 Testování zdola-nahoru (bottom-up testing)

Předpokladem je, že produkt je navržen hierarchicky a hlavní modul volá ostatní moduly na nižší úrovni abstrakce, které volají moduly s ještě nižší úrovní abstrakce, až je zavolán modul, který vykoná požadovanou aktivitu [10]. Při testováním zdola-nahoru se začíná od testování nejnižší úrovně abstrakce až po nejvyšší. Výhodné je použít tuto metodu tehdy, pokud moduly nižší úrovně abstrakce jsou často používány jinými částmi systému.

2.2.5 Testování shora-dolů (top-down testing)

Postup je opačný než při testování zdola-nahoru. Nevýhodou tohoto přístupu je nutnost použití simulace modulů na nižší úrovni.

2.2.6 Regresní testování

Regresní testování je znovu otestování softwaru, na kterém byly vykonány změny, za účelem ověření, že změny nenarušily funkčnost staré verze. Ve firmách, které vyvíjí software s politikou častého vydávání nových verzí(*release early, release often*) je použití regrese nutností. Uživatelé očekávají v nové verzi novou nebo vylepšenou funkcionalitu, ale také si přejí, aby kostra a funkčnost zůstala stejná jako u starší verze. Ověřování, že se v nové verzi neobjevují chyby na místech, které s novou funkčností nesouvisí a ve staré verzi tyto chyby nebyly, se nazývá regrese. Regresní testování může být využito na různých úrovních a vede ke snadnější refaktorizaci a stabilnějšímu vývoji. Často se používají testy na chyby, které byly objeveny ve starších verzích nebo tzv. *smoke testy*, které jednoduše ověří, zda je splněna základní funkčnost produktu. Podle [8] se regresi věnuje 80% času při testování, proto je snahou testy často automatizovat.

2.2.7 Manuální testování

Testování je prováděno lidmi bez pomoci automatizace. Často bývá nahrazeno automatizovaným testováním, v některých případech je ale preferováno ruční testování:

- test vyžaduje použití lidské inteligence,

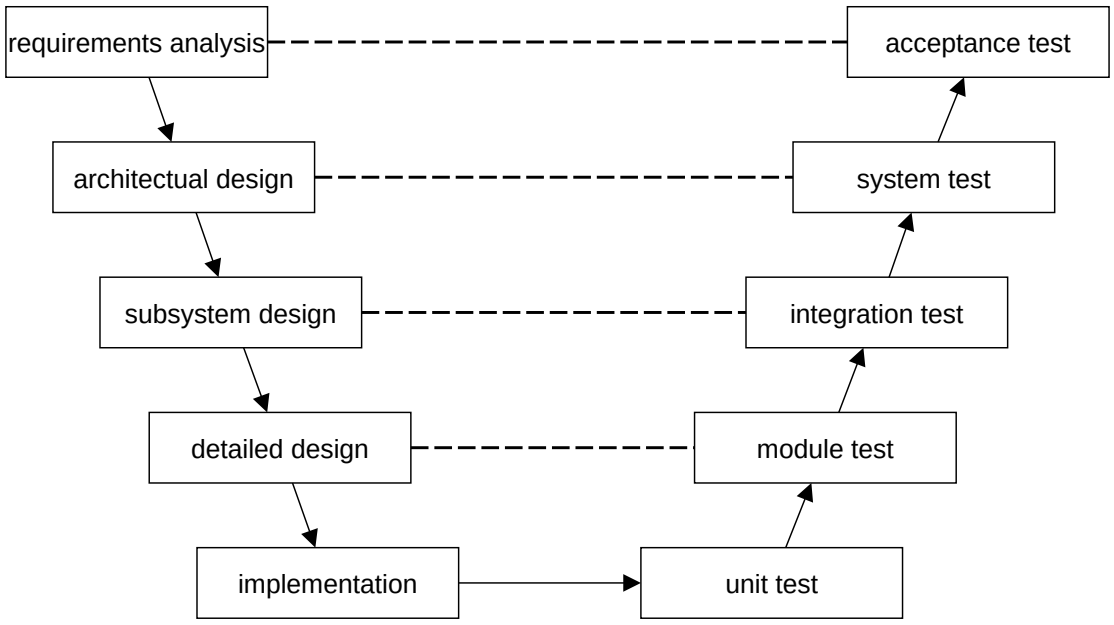
- test zaměřený na grafické zobrazení,
- nastavení prostředí pro test je časově náročné nebo
- nejsou dostupné prostředky pro automatizaci testu

2.2.8 Automatizované testování

Využívání automatizovaného testování může přispět ke snížení časových nároků, minimalizování rizik a zvýšení produktivity a kvality produktu [7]. Před zavedením automatizace by měla proběhnout analýza, zda bude přínosná. Vyhodnocuje se, jak dlouho trvá ruční testování, kolik času automatizace ušetří, jak dlouho bude trvat vytvoření automatizovaného testu, jaké budou časové a finanční náklady pro zavedení automatizace atd. Vytváření automatizovaného testu se skládá ze tří částí. Nejprve se připraví prostředí (*Set Up*) pro testovací případ, čímž se aplikace dostane do počátečního stavu a může se začít s vykonáváním testu. Po vykonání a vyhodnocení testu se provede úklid (*Tear down*) prostředí. Cílem této fáze je dostat prostředí do stavu, v jakém bylo před spuštěním testu. Dochází k mazání vytvořených souborů při testu, odpojování od zdrojů atd.

2.3 Testovací úrovně

Testování je podle [12] rozděleno do pěti úrovní, které souvisejí s úrovněmi při vývoji aplikace. Je doporučováno navrhovat testy současně s fází vývoje, i v případě, že software není ve spustitelné podobě. Vazby mezi testovací a vývojovou fází jsou zobrazeny na obrázku 2.1 jako tzv. *V-model* [4].



Obrázek 2.1: V-model

2.3.1 Jednotkové testování

Nejnižší úrovní testování je jednotkové testování. Jedná se o proces testování individuálních podprogramů v aplikaci. Ověřuje se pouze funkčnost kódu, který je uvnitř jednotky, ostatní volané metody z této jednotky nejsou součástí testování [2]. Testování je zaměřeno na menší základní bloky, kde je snadnější nalézt chybu při ladění. Testování jednotek může probíhat paralelně.

2.3.2 Testování modulu

Testování množiny souvisejících jednotek, které jsou zapouzdřeny do jednoho souboru nebo třídy.

2.3.3 Integrační testování

Při integračním testování dochází k ověření správnosti propojení a komunikace jednotlivých komponent. Předpokladem před tímto testováním je, že jednotlivé moduly pracují správně. Nalezené chyby můžou být příčinou neshody nebo nepochopení požadavků. V roce 1999 selhala sonda vyslaná na Mars, protože na vstup modulu byla místo hodnoty v kilometrech zaslaná hodnota v mílích [4].

2.3.4 Systémové testování

Ověřuje, zda propojením funkčních částí systému funguje jako celek.

2.3.5 Akceptační testování

Akceptačním testováním se ověřuje, zda produkt splňuje předem dohodnuté požadavky. Do testování je zahrnut i uživatel, pro kterého se produkt vyvíjí.

2.4 Testování GUI

Některé uvedené znalosti jsou získané z přednášky předmětu ITS [14]. Vzhledem k tomu, jak je GUI rozšířené a používané, jeho testování je poměrně nový a rozšiřující se obor. Podle [6] obsahuje funkcionalita GUI 45–60% zdrojového kódu. Nejčastější testování GUI ve většině firem vyvíjející software se provádí jako postupné vykonávání kroků s ověřováním správnosti zobrazení a funkčnosti na předem připraveném systému.

Software s GUI obsahuje 2 části: kód aplikační logiky (*business logic*) a GUI, pomocí kterého uživatel komunikuje se systémem. Komunikací je myšleno zasílání signálů, jako jsou například kliknutí na tlačítko, vložení textu, výběr z nabídky aj. Signály jsou odchytávány v systémové logice, která může na tyto uživatelovy aktivity reagovat změnou stavu.

Každý ovládací prvek, který je součástí GUI, může způsobit přechod do jiného stavu systému. Přechod mezi dvěma stavy aplikace může být způsoben mnoha různými vstupními událostmi. Počet permutací sekvencí těchto událostí, což jsou všechny možné požadavky na test, je extrémně vysoký [11]. Pro otestování GUI lze použít některé z kritérií pokrytí:

- Pokrytí události (*Event coverage*)

  Každá událost se musí vykonat minimálně jednou.

- Pokrytí dvojice událostí (*Interaction coverage*)

  Každá možná dvojice událostí se musí vykonat minimálně jednou.

- Pokrytí všech událostí (*UI Event coverage*)

  Při běhu se musí vykonat minimálně jednou všechny možné akce všech ovládacích prvků.

Při testování s využitím vyjmenovaných kritérií pokrytí dojde ke snížení počtu testovacích případů, nicméně tento počet je stále příliš velký a neefektivní v porovnání s úsilím a časem. Pro efektivní otestování GUI bychom potřebovali předem znát vzájemné vazby událostí a stavů testovaného systému. Potřebovali bychom tedy mít informaci o softwaru, který chceme testovat, a to je v rozporu s myšlenkou testování.

Při testování zdrojových kódů je vhodné používat některá z kritérií pokrytí, pro testování funkčnosti GUI nejsou ale tato pokrytí vhodná a je potřeba používat specializované metody a postupy. Kniha [4] rozděluje testování GUI na testování použitelnosti a funkčnosti.

2.4.1 Testování použitelnosti (Usability testing)

Testování použitelnosti má za cíle zjistit, jak je ovládání GUI použitelné, pochopitelné, intuitivní a efektivní. Principem tohoto testování je sledování člověka, který s aplikací pracuje. Nejdříve je potřeba sehnat uživatele, kteří budou aplikaci ovládat. Podle [15] je optimální počet 5-6 osob.

Vybraní uživatelé by se od sebe měli lišit ve věku, vzdělání a ve znalostech informačních technologií, testované aplikace, operačního systému nebo prostředí. Každý uživatel by měl mít svého pozorovatele, který ho bude sledovat při práci. Pro uživatele je připraveno několik scénářů k vykonání. Pozorovatel si u uživatele všímá reakcí, pocitů, myšlenkových pochodů, pohybu očí, myši apod. Pro detailnější zkoumání je možné využít kamerový záznam, který snímá uživatele a testovanou aplikaci. Pokud uživatel nedokáže některý scénář splnit, nebo mu zabral více času než se očekávalo, je analyzováno co uživateli bylo nejasné, jaké očekával chování aplikace a jaké bylo skutečné chování. Na základě těchto výsledků může dojít k upravení vzhledu nebo funkčnosti GUI.

2.4.2 Testování funkčnosti (Functional testing)

Testování funkčnosti je dále děleno na 4 typy:

Ruční testování

Cílem ručního testování je kontrola správného zobrazení, ovládání a dodržování zásad GUI. Mezi nejpoužívanější metody ručního testování patří:

- Náhodné testování

  K náhodnému testování je potřeba tzv. *orákulum*, které specifikuje, jak má vypadat správný běh aplikace. Výsledky náhodného vykonávání akcí u ovládacích prvků jsou porovnávány s orákulem, zda se neporušilo žádné pravidlo testované aplikace. K dostupným nástrojům patří například *EXSYST*.

- Útoky na zabezpečení (Penetration testing)

Útoky na zabezpečení spočívají v tom, že se testeři snaží přemýšlet a postupovat jako hackeři. Při testování chtějí nalézt zranitelná místa, která jsou pak opravena. Toto testování vyžaduje povolení od vlastníka testovaného softwaru, jinak se jedná o hackování, což je nelegální činnost ve většině zemí [1].

- Testování zranitelnosti (Vulnerability scanning)

  Testovací případy jsou vytvářeny na základě databáze známých chyb. Seznam největších rizik zabezpečení softwaru nabízí například *OWASP* [1].

Regresní testování GUI

Princip regresního testování je popsán v kapitole 2.2.6. Pro regresní testování GUI lze využít některý z dostupných nástojů pro nahrávání a spouštění (*capture and replay tool*). Nástroj zachycuje uživatelovu aktivitu a převádí ji do příkazů, které jsou potom vykonány v rámci jednoho testu. Během vykonávání testu jsou pozorovány a vyhodnocovány rozdíly mezi aktuálním a očekávaným stavem aplikace. Nevýhodou tohoto principu je nutnost upravovat testovací případy při grafických změnách ve verzích aplikace. Mezi tyto dostupné nástroje patří například *Sikuli* [2].

Testování validace vstupů

Testuje se, jak aplikace reaguje na nesprávné vstupy. Je očekáváno vypsání informace na chybový výstup nebo upozornění v případě zadání nesprávného vstupu.

Testování míry funkčnosti

Testují se následující požadavky:

- GUI je pohodlné, intuitivní a jednoduché.
- Ovládací prvky pracují tak, jak bylo zamýšleno.
- Systém správně reaguje při nevhodném používání.
- Prozkoumávání, jestli je možné se dostat do všech stavů systému.

Testování pomocí rozpoznávání objektů

Po spuštění testované aplikace se na obrazovce hledají vzorce a simulují se uživatelské vstupy. Pro rozpoznávání obrazu lze použít knihovnu *OpenCV*. Dále se využívá dostupných nástrojů pro rozpoznávání textu. Mezi nástroje využívající technologii rozpoznávání objektů řadíme *Sikuli*, *Xpresser* aj. Výhodou používání této technologie je, že se testuje právě to, co uživatel vidí. Naopak mezi nevýhody patří:

- Zpracování obrazu a rozpoznávání je časově náročné.
- Nepřesnosti při rozpoznávání(odlišný styl, typ písma, rozlišení).

  Částečným řešením je používání jednotného prostředí.
- Více podobně vypadajících nebo stejných objektů.

  Problém lze obejít omezením plochy pro vyhledávání.

[1] https://www.owasp.org
[2] http://lab.sikuli.org/

Testování se znalostí objektů

Využívají se nástroje, které nahlížejí do interních struktur ovládacích prvků. Ty se napojují na ovladač frameworku, který GUI vykresluje. Nepracuje se s aktuálním obrazem plochy testované aplikace, ale s hierarchicky uspořádanými objekty, které jsou uloženy v interní struktuře. Při testování webových aplikací je interní strukturou myšlen například objektový model dokumentu (*document object model*). Z těchto struktur lze získávat informace o velikosti, stylu nebo stavu ovládacího prvku. Mezi dostupné nástroje patří *Selenium* (web), *Dogtail* (GTK+) nebo *Squish* (Qt).

- Výhodou testování se znalosti objektů je:

  rychlost ovládání a

  nezávislost na stylu GUI.

- Nevýhodou je, že metoda nedokáže:

  ověřit grafické zobrazení objektu(špatné zarovnání, přetékání) a

  rozpoznat špatně viditelné objekty(objekt překrytý oknem).

Kapitola 3

Specifikace požadavků pro využívání knihovny

Pro správné využití knihovny musí být splněny vyjmenované požadavky.

3.1 Požadavky na rozhraní knihovny

Rozhraní knihovny obsahuje následující funkčnost:

- rozpoznání objektů GUI,
- ovládání objektů,
- manipulace s myší a klávesnicí,
- zasílání klávesových zkratek,
- ovládání virtualizovaného počítače (zapnutí, vypnutí, snapshot, vykonání příkazu),
- získávání snímku plochy virtualizovaného počítače.

3.2 Požadavky na testovanou aplikaci

Testovaná aplikace musí splňovat následující požadavky:

- je jedno-okenní aplikace,
- tooltip objektu musí být zobrazen, dokud kurzor myši neopustí hranici tohoto objektu [1],
- tooltip při posunu myši uvnitř objektu nemění svoji pozici.

3.3 Požadavky na virtualizaci

Důvodem pro testování na virtualizovaném stroji je efektivní možnost využívání různých operačních systémů. Následky selhání virtuálního stroje nemusí způsobovat škody na hardwaru fyzického počítače.

[1] `Firefox nesplňuje tento požadavek, bug je nahlášen na` https://bugzilla.mozilla.org/show_bug.cgi?id=82953.

3.4 Požadavky na ukládání dat

Vyhledané objekty testované aplikace lze uložit do XML. Pro urychlení rozpoznávání bude možné načítat tyto konfigurace.

Kapitola 4

Návrh aplikační vrstvy knihovny pro testování GUI

V kapitole se seznámíme s návrhem knihovny pro testování GUI. Celá knihovna je rozdělena do několika modulů, jejichž funkčnost a základní vlastnosti jsou uvedeny v první části kapitoly. Druhá část popisuje princip komunikace mezi jednotlivými moduly.

4.1 Moduly knihovny

V kapitole jsou uvedeny rozhraní, které jednotlivé moduly nabízí. Také jsou zmíněny principy modulů a technologie, které lze k jejich funkčnosti využít.

4.1.1 VNC klient

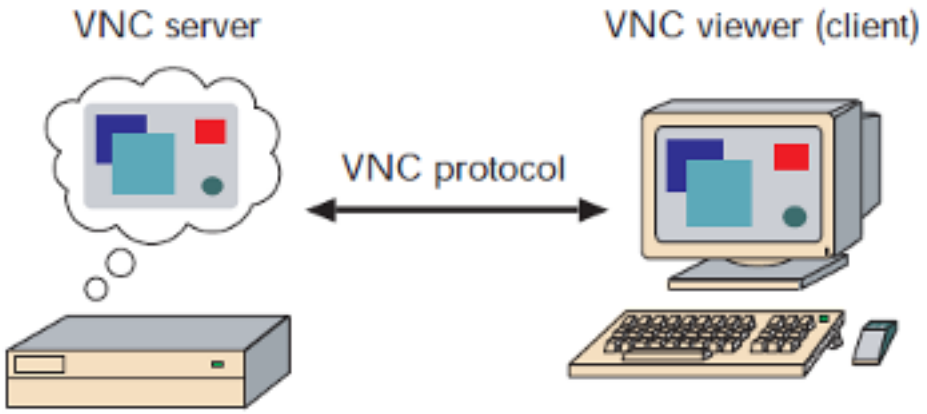
Detekce GUI komponent může být rozeznána na základě toho, jak objekt reaguje na uživatelské aktivity s klávesnicí a myší. Modul bude využívaný při rozpoznávání GUI komponent tímto algoritmem. Další funkcí bude získávání screenshotu plochy, z kterého se budou GUI objekty detekovat. Mezi základní funkčnost knihovny patří manipulace s nalezenými objekty v GUI pomocí klávesnice a myši. Modul se připojí k virtuálnímu počítači, který je vyhledán podle IP adresy a čísla portu. Poté simuluje typické uživatelské aktivity (pohybování a klikání myší, stisknutí či uvolnění klávesy, zaslání klávesové zkratky aj). Modul bude využívat technologii *VNC*.

- **VNC (Virtual Network Computing)**

  VNC umožňuje uživateli připojení ke grafickému výstupu programu pomocí počítačové sítě. VNC pracuje na principu klient-server. Server posílá aktuální podobu grafické plochy počítače klientovi, který ji zobrazuje uživateli. K serveru může být připojeno více klientů, kteří mohou plochu sdílet. Jelikož je přes síť zasílán pouze aktuální obraz plochy, je VNC protokol nezávislý na operačním systému.

  Pro ustanovení spojení mezi klientem a serverem nejprve server žádá klienta o autentizaci, která většinou zahrnuje zadání hesla. Dále se obě strany dohodnou na komunikaci (rozměry plochy, formát pixelu, kódování). Nakonec klient požádá o aktualizaci celé obrazovky [13]. Když chce klient obnovit obraz plochy, zašle požadavek na server. Pokud uživatel interaguje s myší nebo klávesnicí, jsou tyto události zaslány na server. Modul VNC klienta nevyžaduje lidskou přítomnost. Zprávy jsou vytvořené

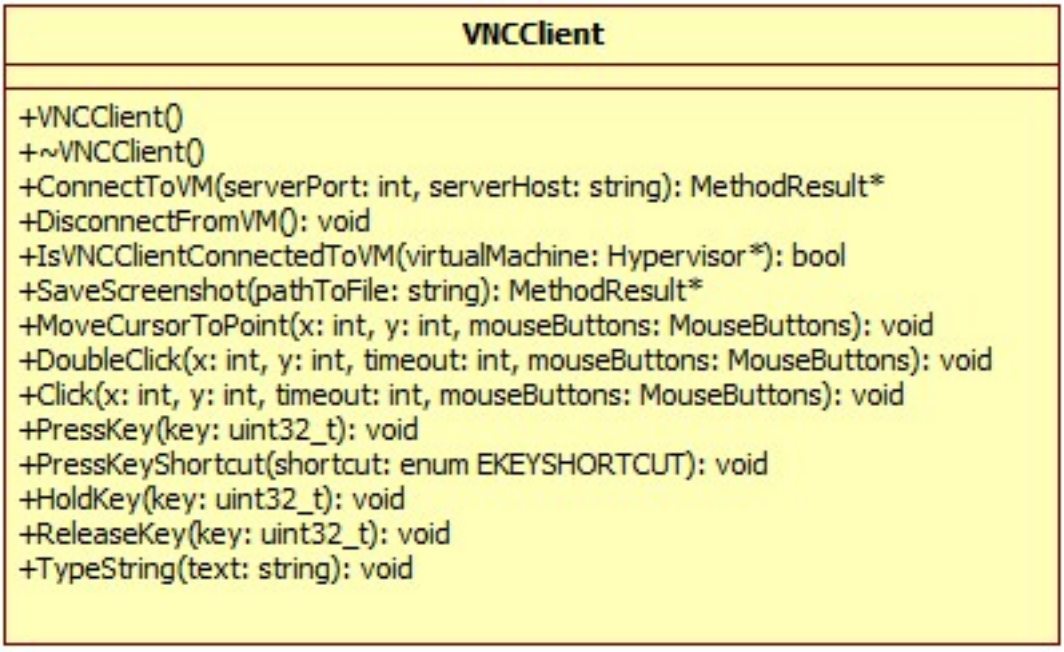
podle potřeb knihovny a poslány na server ve stejném formátu, jako kdyby tuto aktivitu vykonal člověk pomocí manipulace s klávesnicí nebo myší. Server ani netuší, že ve skutečnosti nedošlo k žádné fyzické aktivitě. Komunikace probíhá přes protokol *RFB (remote framebuffer)*[1], který používá jako výchozí TCP port 5900. Výhoda RFB protokolu spočívá v jednoduchosti a rozšiřitelnosti.



Obrázek 4.1: VNC technologie, obrázek převzat z [13]

Podobným protokolem je *RDP (Remote Desktop Protocol)*. Hlavní rozdíl mezi RDP a RFB je způsob přenášení informací o grafickém výstupu mezi klientem a serverem. U RDP protokolu se neposílá obraz plochy, posílají se pouze instrukce, jakým způsobem má klient plochu vykreslit. Vzhled vykreslené plochy proto záleží na platformě a může se lišit. Existuje mnoho VNC klientů pro různé platformy: *RealVNC*, *TightVNC*, *ThinLinc*, *TigerVNC* atd.

[1] http://www.realvnc.com/docs/rfbproto.pdf



Obrázek 4.2: UML Diagram tříd modulu VNC klient

4.1.2 Rozpoznávání obrazu

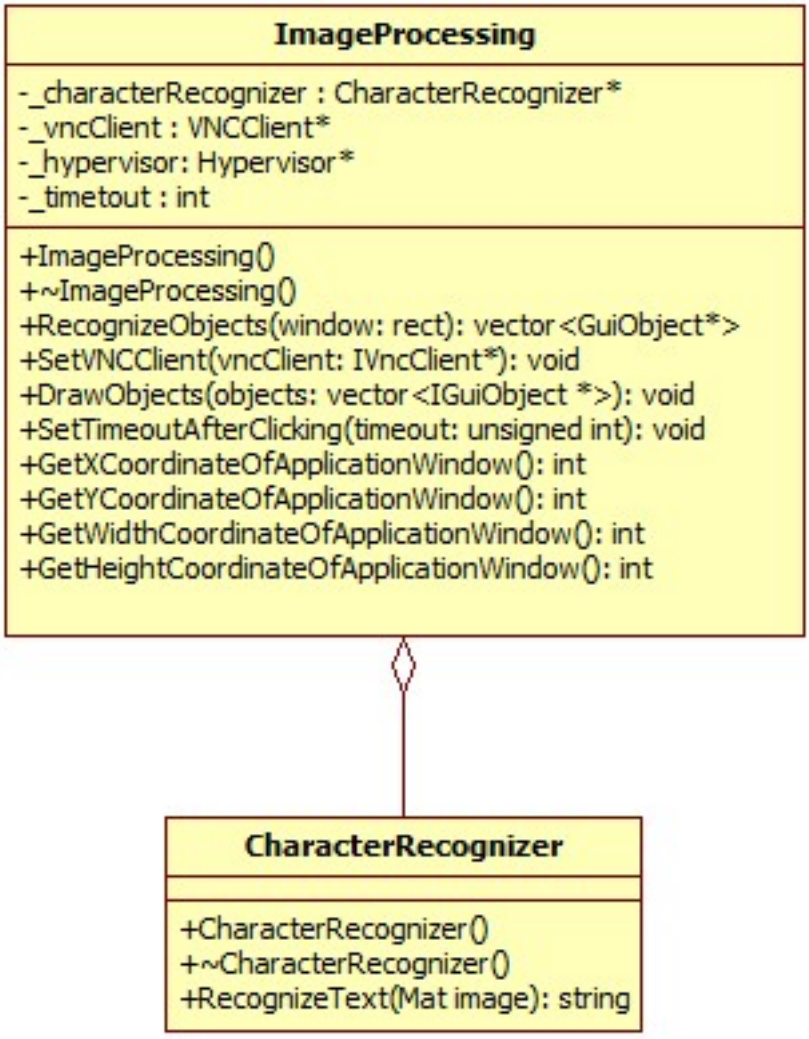
Modul rozpoznávání obrazu je nejdůležitější částí knihovny a jejím cílem je detekování GUI komponent. Při rozpoznávání využívá modul hypervizor pro vytváření a obnovování snapshotů, optické rozpoznání znaků pro získávání informací o zkoumaném objektu a VNC klienta na simulování lidských aktivit pro detekování objektů na základě grafických změn během manipulace s nimi. Funkčnost modulu pro rozpoznávání obrazu lze rozdělit do dvou hlavních částí:

1. Detekce objektů v obraze
2. Porovnání dvou obrazů

Jednotlivé komponenty GUI lze vyhledávat podle jejich typických vlastností. Testovaná aplikace je dopředu neznámá a nemáme představu, jaké objekty a jakých tvarů bude obsahovat. Proto je nutné aplikovat na detekci komponent různá omezení, například maximální rozměry objektu, maximální délka textu uvnitř tlačítka atd. Při rozpoznávání komponent se bude využívat porovnávání změn mezi obrazy a následné vyhodnocování těchto rozdílů. Základní myšlenkou je, že objekt reaguje svým vlastním typickým způsobem na události přicházející od klávesnice a myši. Při kliknutí nebo přejetí kurzoru myši nad tlačítkem může dojít ke změně stylu tlačítka, obsah pole pro zadání textu se změní po stisknutí tlačítek, hypertextové odkazy se podtrhnou při přejetí kurzoru myši atd. Rozpoznáváním rozdílů před vykonanou akcí a po ní, lze vyvodit vlastnosti daného objektu a určit tak jeho nejpravděpodobnější typ. Tímto srovnáním také odvodíme, jakou oblast komponenta ovládá a jak

se změnilo GUI. Po vykonané akci se můžou objevit nové komponenty, existující objekty mohou změnit vlastnosti (změna textu, pozice aj.) nebo být úplně odstraněny. Požadované funkčnosti můžeme dosáhnout vhodnou kombinací následujících algoritmů:

- histogram,
- porovnávání vzoru s šablonou (*template matching*),
- detekce hran, obdélníkových tvarů, přímek, rohů aj.,
- deskriptor oblasti,
- prahování,
- shluková analýza,
- spojování oblastí aj.



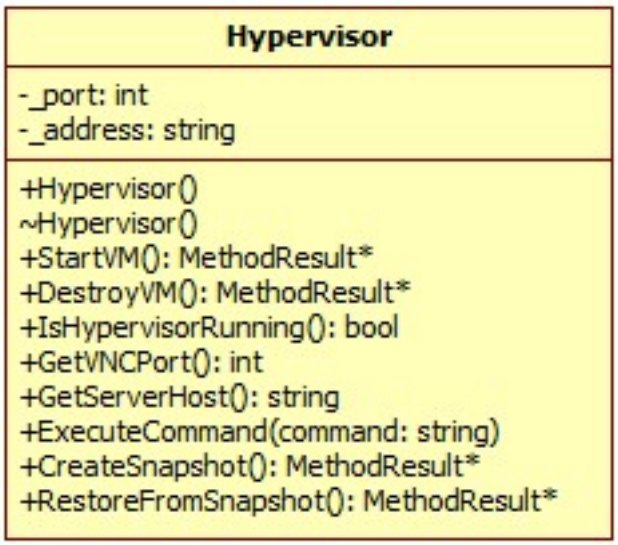
Obrázek 4.3: UML Diagram tříd modulu Rozpoznávání obrazu

4.1.3 Hypervizor

Jedním z požadavků v kapitole 3 bylo použití virtualizace. Jedná se o vytvoření virtuálního počítače, který se skládá z virtuálních komponent (virtuální paměť, procesor, disk a další periferie). Mezi základní techniky virtualizace patří **Plná virtualizace Hardwarová virtualizace** a **Paravirtualizace** [9]. Protože knihovna využívá VNC technologie, musí být spuštěný virtuální počítač přístupný pro VNC klienta. Port a adresa pro připojení přes VNC jsou specifikovány při spuštění virtuálního stroje. Modul hypervizor bude nabízet následující prostředky pro základní ovládání a manipulaci s virtuálním počítačem:

- Spuštění a vypnutí
- Vykonání příkazu na virtuálním stroji

- Vytváření snapshotů a jejich obnovování



Obrázek 4.4: UML Diagram tříd modulu Hypervizor

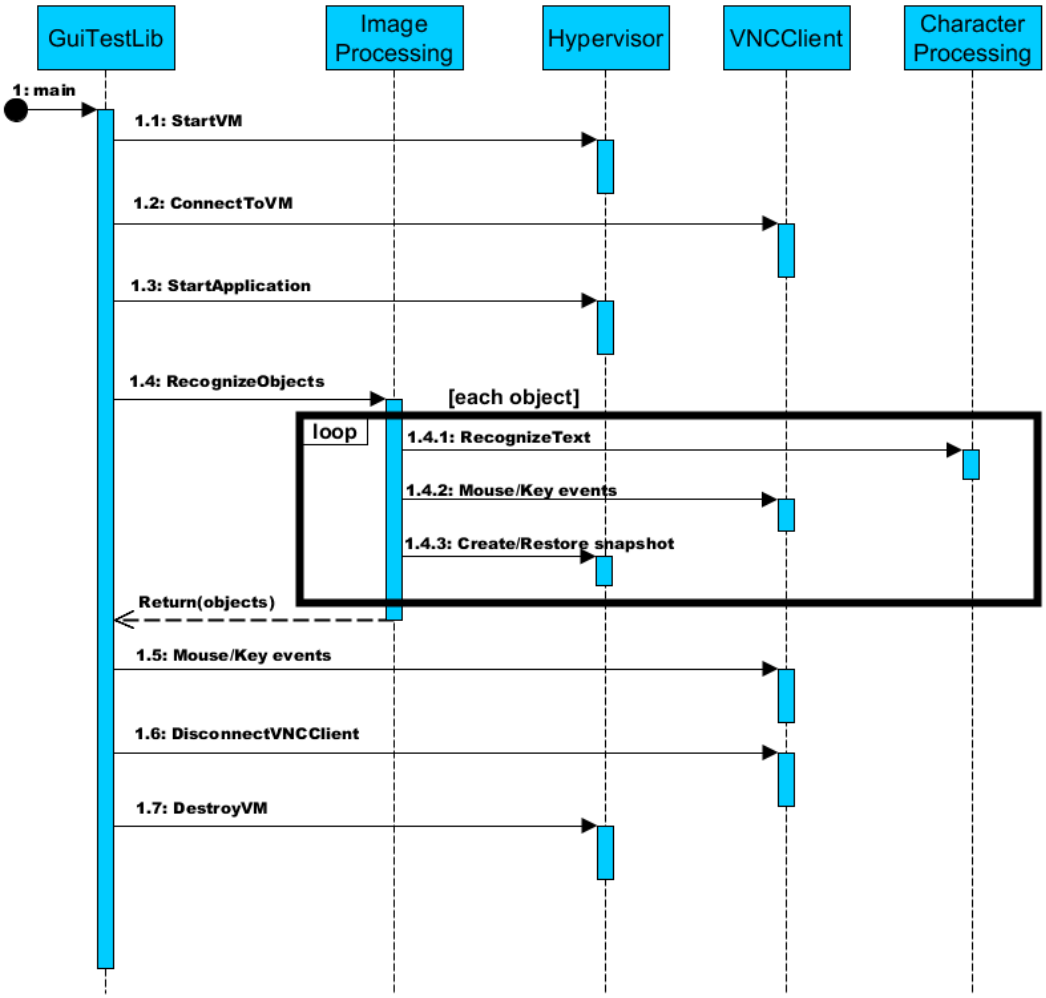
4.1.4 Optické rozpoznávání znaků

Modul optického rozpoznávání znaků je součástí modulu pro rozpoznávání obrazu. Vstupem bude obraz, ze kterého se modul bude snažit rozpoznat text. Čím větší obraz bude, tím více bude rozpoznaný text obsahovat nepřesností. Pro minimalizování obrazu se používá algoritmus pro vyhledání textových oblastí. Pouze tato textová oblast z obrazu je použita jako vstup pro rozpoznání textu. Rozpoznávání textu se bude potřebovat v rámci práce s nějakým GUI objektem, například rozpoznání textového popisu u tlačítka. Mezi nástroje pro rozpoznávání textu patří: *Tesseract*, *SimpleOCR*, *GOCR*, *FreeOCR*.

4.2 Činnost knihovny

Knihovna pro automatizované rozpoznávání GUI se jmenuje **GuiTestLib** a její rozhraní je definováno v kapitole 5.3. Uživatel při práci s knihovnou používá pouze rozhraní modulu GuiTestLib, který zapouzdřuje všechny zmíněné moduly v předchozí kapitole. Pro využívání metod knihovny se nejprve musí spustit virtuální počítač a připojit se na něho pomocí VNC klienta. Poté se na virtuálním počítači spustí testovaná aplikace. Pro testování aplikace je potřeba znát rozmístění, vlastnosti a typy objektů. Proto se detekují všechny oblasti, které s velkou pravděpodobností obsahují nějaký GUI objekt. Tyto oblasti (dále značené jako *windows*), jsou určeny pomocí algoritmů pro zpracování obrazu. Při rozpoznávání se může použít ovládání kurzoru myši, na kterou objekt může reagovat změnou některých vlastností, z čehož můžeme dále určit rozměry tohoto objektu. V další fázi je třeba u každého window klasifikovat, o jaký GUI element se jedná a jaké má vlastnosti. Při roztřídění se využívá

simulace klávesnice a myši. Z vypozorovaných příznaků se objektu přiřadí nejpravděpodobnější typ a odvodí se jeho vlastnosti. Při určení vlastností se například využije rozpoznávání textu z obrázku. Při zkoumání GUI objektů tímto způsobem se může změnit stav testované aplikace. Z tohoto důvodu se bude pořizovat snapshot a v případě potřeby se obnoví, čímž se aplikace dostane do stejného stavu a bude se moci pokračovat klasifikováním dalších windows. Výsledkem této fáze bude seznam objektů testované aplikace. Objekty a jejich vlastnosti se dají uložit do XML souboru. Protože jednou z nevýhod při rozpoznávání obrazu je časová složitost, je dobré u často testované aplikace využívat možnost načtení scény z XML souboru. Když jsou všechny objekty rozpoznané, uživatel s nimi může manipulovat a ověřovat tím některý testovací případ. Po skončení testování se odpojí VNC klient od virtuálního počítače, poté se tento počítač vypne. Princip knihovny a komunikace mezi jednotlivými moduly je znázorněna na obrázku 4.5.



Obrázek 4.5: Sekvenční diagram knihovny pro testování GUI

Kapitola 5

Implementace aplikační vrstvy pro testování GUI

Knihovna je nazvána **GuiTestLib** a pro její implementaci byl použit jazyk C++. Operačním systémem použitým při vývoji bylo *Ubuntu 12.04 LTS*. V této kapitole jsou popsané použité technologie při implementaci jednotlivých modulů, základní třídy, algoritmy pro rozpoznávání objektů, formát XML pro uložení a problémy, které se vyskytly při implementaci.

5.1 Základní třídy knihovny

Knihovna pracuje s tvary všech detekovaných objektů jako s obdélníky, které jsou popsány třídou `Rect` z knihovny OpenCV. Vlastnosti `x` a `y` určují pozici levého horního rohu.

```
Rect(){
  int x;
  int y;
  int height;
  int width;
}
```

GuiTestLib při svém běhu volá metody jednotlivých modulů, jejich výsledek je vrácen ve třídě `MethodResult`. Vlastnost `result` určuje jestli metoda selhala nebo proběhla úspěšně.

```
MethodResult {
  bool result;
  string errorMessage;
  string infoMessage;
}
```

Pokud metoda modulu selže, třída `MethodResult` obsahuje v `errorMessage` chybovou zprávu, která je převzata z chybového kódu. Chybové kódy jsou specifikovány v třídě `TestingGUIErrors`. Protože při testování se většinou rozlišuje to, zda testovací případ prošel nebo ne, nemá smysl pokračovat v testu, který už selhal v některé své části a knihovna vyhodí výjimku. Každý modul má definovaný svůj typ výjimky, kterou může při vyskytnutí

chyby vyhodit. Při používání metod, které nejsou zcela implementovány (viz kapitola 5.7) je vyhozena výjimka `EXC_NOT_IMPLEMENTED`.

Pomocné metody pro jednotlivé moduly jsou umístěny ve složce `helpers`. Knihovna používá mnoho konstant, které byly odvozeny z pozorování a typického chování GUI většiny aplikací. Všechny konstanty jsou definovány v hlavičkovém souboru `constants.h`

5.2 Použité technologie

V kapitole jsou uvedeny nástroje, které byly využity pro implementování funkčnosti modulu.

5.2.1 Libvirt

Pro implementaci modulu Hypervizor byl využitý nástroj *libvirt 1.2.3* [1] vyvíjen společností *Red Hat*. Virtualizovaný stroj byl vytvořen programem *QEMU 2.0.0*. Libvirt se snaží sjednotit ovládání virtualizovaných platforem a podporuje nejpoužívanější virtualizační nástroje: *KVM/QEMU*, *VirtualBox*, *Xen*, *Microsoft Hyper-V* a další [3]. Informace o virtualizovaném systému, jako jsou například velikost paměti, použitá virtualizační technologie nebo možnost připojení přes VNC, jsou specifikovány v XML formátu. Libvirt označuje virtualizovaný systém jako tzv. *doménu* a její XML formát pro konfiguraci s příklady použití je uveden na webových stránkách libvirtu [2]. Při zasílání signálů od VNC klienta pro pohyb myši na virtualizovanému stroji docházelo k posunu myši na jinou pozici, než bylo požadováno. Tento nedostatek byl vyřešen přídáním tagu `input` s typem `tablet` do konfiguračnho XML dokumentu:

```
<domain>
  <...>
  <device>
    <input type='tablet' bus='usb'/>
  <devices/>
  <. . .>
</domain>
```

5.2.2 LibVNCClient

Funkčnost modulu VNC klienta zajišťuje *LibVNCServer/LibVNCClient 0.9.9* [3]. Knihovna nabízí také třídu pro vytvoření VNC serveru, ke kterému se klient může připojit. V GuiTestLib se ale třída LibVNCServer nepoužívá, protože VNC server je inicializován při spuštění virtuálního počítače.

5.2.3 OpenCV

Nedílnou částí funkčnosti knihovny je zpracování obrazu, o který se stará *OpenCV 2.4.6.1* [4]. OpenCV je open-source knihovna zaměřená na počítačové vidění a zpracování obrazu a může běžet pod platformami Linux, Windows a Mac OS X. Je implementována v C a C++, lze ji využít s rozhraním pro Python, Ruby, Matlab a další programovací jazyky [5].

[1] http://libvirt.org/
[2] http://libvirt.org/formatdomain.html/
[3] http://libvnc.github.io/
[4] http://opencv.org//

5.2.4 Tesseract

Při vyhledávání a klasifikaci GUI elementů je zapotřebí nástroj pro optické rozpoznávání znaků (*Optical Character Recognization)*. Pro tento účel byl použit *Tesseract 3.02.02*. Snahou je předávat nástroji co nejmenší obrázek s textem, aby byl počet chyb při rozpoznání minimální. Případné chyby při rozpoznávání textu můžou ovlivnit průběh testovacího případu.

5.3 Rozhraní knihovny

Všechny detekované objekty jsou k dispozici v proměnné `actualScene` a jsou zděděny od třídy `IGuiObject`. Rozhraní knihovny je následující:

```
GuiTestLib();
virtual ~GuiTestLib();
MethodResult* StartVM(IHypervisor *hypervisor);
MethodResult* StartVMAndWaitForBooting(IHypervisor *hypervisor, int delay);
MethodResult* ConnectVNCClientToRunningVM(IVncClient *vncClient);
void DisconnectVNCClientFromVM();
MethodResult* DestroyVM();
void RecognizeObjects();
void SaveScreenshot(string pathToFile);
void SaveActualSceneToXML(string pathToXmlFile);
void LoadActualSceneFromXML(string pathToXmlFile);
void StartApplication(void(*f)());
void StartApplication(string command);
void MouseMoveCursorToPoint(int x,int y,mouseButtons mouseButtons);
void MouseDoubleClick(int x, int y,mouseButtons mouseButtons);
void MouseClick(int x, int y,mouseButtons mouseButtons);
void KeyboardPressKey(uint32_t key);
void KeyboardHoldKey(uint32_t key);
void KeyboardReleaseKey(uint32_t key);
void KeyboardTypeString(string string);
void KeyboardPressKeyShortcut(enum EKEYSHORTCUT shortcut);
bool IsVNCClientConnectedToVM();
void ShowObjects();
void CreateSnapshot();
void RestoreFromSnapshot();
```

5.4 Vyhledávání oblastí v GUI

Nejdůležitější funkčností knihovny GuiTestLib je rozpoznání všech GUI objektů a jejich vlastností. Před rozpoznáváním nemáme žádné informace o tom, jak bude testovaná aplikace vypadat a jaké bude obsahovat objekty. Vyhledání a rozpoznání všech objektů pouze na základě screenshotu aplikace je velmi složité a nepřesné. Z tohoto důvodu se nejprve naleznou všechna windows, ze kterých se pak v další fázi budou klasifikovat GUI objekty.

Detekce windows je zajištěna třemi různými přístupy: vyhledávání textových windows, obdélníkových windows a windows, které mění vzhled po přejetí kurzoru myši. Výhody, nevýhody a principy jednotlivých algoritmů jsou vypsány dále v této kapitole. Pokud se windows detekovány různými metody překrývají, ponechá se pouze window vyhledané metodou s největší prioritou (v případě malého překrývání se může oblast oříznout). Nejpřesnějším rozpoznání je vyhledávání na základě grafické změny objektu při přejetí myši. Tato metoda má nejvyšší prioritu, následována vyhledáváním obdélníkových oblastí. Nejnižší prioritu má metoda pro detekování textu, která je ale později využívána pro získávání informací o objektu, jako je například textový popis tlačítka.

U větších textových oblastí se může stát, že některá slova nebo i písmena budou falešně detekovány jako obdélníkové oblasti. V dalším průchodu bude rozpoznáno, o jaký GUI objekt se jedná a pokud nelze oblasti určit žádný typ GUI objektu, toto window bude odstraněno a metoda pro detekci textu už úspěšně detekuje tento text.

5.4.1 Oblasti měnící vzhled po přejetí kurzoru myši

Kurzor myši je umístěn do levého horního rohu aplikace[5] a je posouván k jejímu pravému okraji. Poté je kurzor posunut směrem dolů a akce se opakuje, dokud se neprověří celá scéna aplikace. Délka posunutí kurzoru je definována v `constant.h` a byla stanovena na 10 pixelů. Při menší hodnotě by se časová náročnost rozpoznávání zvýšila, při větším posunu by mohlo dojít k nedetekování GUI elementů s rozměrem menším než stanovená hodnota. Po každém posunu se čeká vteřinu, zda se neobjeví nějaká grafická změna v aplikaci. Během této akce může nastat několik grafických změn, které vůbec nesouvisí s elementem, nad kterým je kurzor myši. Může se jednat o:

1. změnu mimo aplikaci,
2. blikání kurzoru, animace, spuštěné video, banner, změna času nebo
3. změnu ikony kurzoru myši.

Změny, které se objeví mimo testovanou aplikaci, jsou ihned ignorovány. Před začátkem této metody se analyzuje scéna a hledají se v ní změny, které se dějí nezávisle na uživatelově aktivitě. Při analýze se nemanipuluje s klávesnicí ani myší, pouze se pozorují grafické změny, které vznikly bez jakéhokoliv uživatelova zásahu. Pokud se tyto změny nachází uvnitř aplikace, jedná se o windows, které budou v dalším běhu klasifikovány. Při prozkoumávání ostatních míst se ale budou podobné změny ignorovat.

Při pohybování se může ikona kurzoru změnit a detekování změny může být nežádoucí. Proto se vždy ignorují změny malých velikostí, které se objevily v blízkosti pozice kurzoru před a po hnutí kurzoru. Těmito kroky se odstraní nežádoucí změny, které nijak nesouvisí s elementem pod kurzorem myši. Pokud zůstane alespoň jedna změna, znamená to, že v místě kurzoru je nějaký objekt, který na kurzor reaguje. V tu chvíli se volá metoda `FindElementBorder()`, která hledá hranice objektu. Hranicí je místo, kde se vzhled objektu mění na vzhled, který měl objekt, když nad ním nebyl kurzor.

Problematická je detekce sousedících objektů měnící svůj vzhled. Po detekci prvního objektu se kurzor myši posune za jeho pravou hranici, to je dovnitř sousedícího, a pokračuje se dále ve vyhledávání. V tomto okamžiku už kurzor myši je nad objektem, který změnil vzhled. Algoritmus detekuje změnu až tehdy, kdy kurzor myši opustí tento objekt. Řešením

[5] souřadnice levého rohu jsou x=0,y=0

tohoto problému je vyhledání míst, které neobsahují žádné obrysy. Po detekování hranic objektu se kurzor myši přesune na jedno z těchto prázdných míst, až poté je pokračováno dále v rozpoznávání. V případě sousedícího objektu je změna ihned detekována.

Algoritmus je kvůli časové náročnosti optimalizován. Při posouváni kurzoru jsou přeskakovány prázdná a detekovaná místa. Metoda je spolehlivá pro detekci tlačítek, hypertextových odkazů a objektů s tooltipem. V kapitole 3 jsou definovány požadavky, které tooltip musí splňovat, jinak algoritmus selhává. Při vyhledávání hranic se kurzor myši dostane z objektu, poté je potřeba se s kurzorem vrátit na stejné místo jako na začátku prozkoumávání, aby byl případný tooltip zobrazen na stejné pozici.

5.4.2 Nalezení obdélníkových oblastí

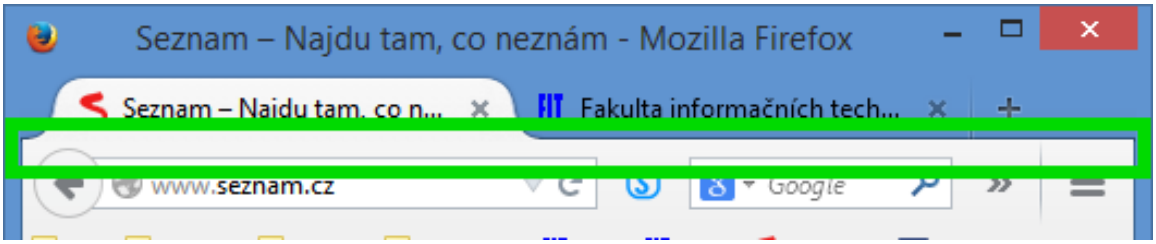
Vstupem je obrázek aplikace, ze kterého se metoda snaží rozpoznat všechny obdélníkové oblasti. Nejprve se detekují hrany pomocí *Cannyho algoritmu*, jeho popsaný princip je převzat z [5]. Algoritmus nejdříve najde potenciální pixelové hrany a u každé se porovná její gradient s horním a dolním prahem[6]. Pokud je gradient větší než horní práh, pixel je označen za hranu. Když je gradient mezi dolním a horním prahem, pixel je označen za hranu pouze v případě, že sousedí s pixelem, který má gradient větší než horní práh. V jiném případě pixel není prohlášen za hranu. Výstupem Cannyho metody je seznam hran, tzv. kontur (*contours*).

Každá kontura je pak aproximována, čímž se počet jejích bodů sníží a tvar kontury se zjednoduší. Zda kontura znázorňuje nějaký obdélníkový objekt, se zjistí porovnáním obsahu uzavřené kontury s obsahem obdélníku, který konturu opisuje. Problém nastává, když je obdélníkový objekt popsán více než jednou konturou (například dvěma konturami v tvaru písmene "L"). Při tomto algoritmu je každá kontura zahozena, protože nemá obdélníkový tvar. Pro detekci obdélníkových oblastí, které jsou popsány více konturami, se použije algoritmus pro nalezení konvexní obálky (*convex hull*).

Při používání Cannyho algoritmu může dojít k nepřesnostem a hranice obdélníku nemusí být detekována přesně. Metoda `AlignWindows` vyhledá obdélníky překrývající se pouze malou částí a zarovná je tak, aby spolu sousedily a nepřekrývaly se. Ze všech obdélníkových oblastí jsou ještě vymazány ty, které opisují alespoň jeden menší obdélník. Tím jsou odstraněny oblasti, které rozdělují GUI do určité hierarchie. Může se jednat například o toolbar oblast, která zapouzdřuje obdélníkové ikony.

U každého window se ještě ověří, zda jeho rozměry odpovídají omezením (minimální výška nebo šířka atd.). Výstupem metody je seznam všech obdélníkových oblastí nacházející se v obrázku. Algoritmus je ideální pro detekci tlačítek, naopak nedokáže detekovat záložky. Většina záložek totiž nemá přesně obdélníkový tvar, jejich dolní hranice je rozšířena na šířku všech záložek. Tento problém je zvýrazněn na obrázku 5.1. Záložky jsou ale detekovány algoritmem vyhledávání oblastí měnící vzhled po přejetí kurzoru myši.

[6] Horní a dolní práh je doporučeno zvolit mezi poměry 2:1 a 3:1



Obrázek 5.1: Zvýrazněná dolní hranice záložky v prohlížeči Firefox

5.4.3 Nalezení textových oblastí

Pro nalezení textových oblastí se také využije Cannyho detektor. Odstraní se kontury s velkou výškou a šířkou, které pravděpodobně znázorňují hranici nějakého tlačítka nebo oblasti. Ostatní kontury jsou slučovány pokud se překrývají či zapouzdřují nebo pokud jsou splněny následující podmínky:

1. kontury mají alespoň jednu stejnou y-ovou souřadnic,
2. vzdálenost x-ové souřadnice není větší než stanovená hodnota,
3. výška kontur je v určitém poměru a
4. y-ový rozdíl souřadnic spodních hranic kontur nepřesahuje stanovenou hodnotu.

Tímto postupem jsou sloučeny jednotlivé kontury popisující části písmen do obdélníkové oblasti, která obsahuje větší souvislý text. Problémem je určit konstantu pro maximální x-ovou vzdálenost mezi dvěma konturami. Důležité je, aby tato hodnota byla menší než vzdálenost mezi dvěma položkami v textovém menu. Jinak by došlo ke sloučení celého textového menu a následné klasifikování tohoto window by bylo složité. Z tohoto důvodu je konstanta definována na 7 pixelů. Nevýhodou je, že tímto může být souvislý text rozdělen na windows, které obsahují pouze jedno slovo. Při klasifikaci se ale zjistí, že se jedná pouze o textovou oblast a může být sloučena s ostatními slovy.

Dalším problémem jsou interpunkční znaménka. Znaky jako ”. - ’_ ”nejsou kvůli své výšce sloučeny s textem, ke kterému se vztahují. Proto je v dalším běhu u každého window s textem zkoumáno, zda v malé oblasti od pravé strany není nějaká kontura. Při této fázi se rozšíří některé windows o interpunkci. Protože se touto fází mohly změnit rozměry některých oblastí, znovu se provede algoritmus pro sloučení textových oblastí.

5.5 Rozpoznávání GUI objektů

V další fázi jsou postupně prozkoumávány a testovány všechny windows. Cílem je rozpoznat, jaký GUI objekt a s jakými vlastnosti oblast obsahuje. Pro klasifikování oblasti se využívají příznaky jako jsou:

1. velikost elementu,
2. algoritmus, který window detekoval,
3. změny v testované aplikaci při stisknutí tlačítka myši nebo klávesnice,

4. změny pouze uvnitř window při stisknutí tlačítka myši nebo klávesnice,

5. přítomnost elementu podobného tvaru jako přepínač nebo checkbox aj.

Při rozpoznávání je důležité, jakým algoritmem z kapitoly 5.4 byl window detekován. Například když se při stisknutí tlačítka myši nezmění nic v testované aplikaci a oblast byla detekována jako textová, vyhodnotí se obsah window jako textový popisek. Pokud se oblast mění při přejetí kurzoru myši, ale po stisknutí tlačítka se nic nezmění, předpokládáme, že se jedná o neaktivní tlačítko, kterému se při přejetí myši zobrazí tooltip. Vstupem metody `GetGuiObjectsFromWindows` jsou všechny windows, výstupem pak GUI elementy s vlastnostmi. Objekty, které lze detekovat v testované aplikaci jsou zděděné ze třídy `IGuiObject`. Objekty jsou definovány i s jejich vlastnostmi a možnostmi pro manipulaci v souboru `guiobjects.h`. Při rozpoznávání bereme v úvahu to, že některé oblasti mohly být falešně detekovány a neohraničují žádný objekt. Při klasifikaci se manipuluje s myší a klávesnicí, čímž může dojít k změně stavu aplikace. Proto se využívá vytváření a obnovování snapshotů.

5.6 Reprezentace dat ve formátu XML

Proces vyhledávání a rozpoznávání objektů je časově náročný, proto se vyplatí rozpoznané objekty ve scéně ukládat a při příštím testování načítat. Pro práci s XML soubory je použita knihovna *Libxml2 2.7.8*. Popis struktury XML dokumentu pomocí *DTD* (*Document Type Definition*) pro definované objekty je následující:

```
<!DOCTYPE ApplicationWindow [
  <!ELEMENT ApplicationWindow (GuiElement)+>
  <!ELEMENT GuiElement (Button|TextInput|Checkbox|Label|UnknownObject)>
  <!ELEMENT Button EMPTY>
  <!ELEMENT TextInput EMPTY>
  <!ELEMENT Checkbox EMPTY>
  <!ELEMENT Label EMPTY>
  <!ELEMENT UnknownObject EMPTY>

  <!ATTLIST ApplicationWindow x CDATA #REQUIRED>
  <!ATTLIST ApplicationWindow y CDATA #REQUIRED>
  <!ATTLIST ApplicationWindow width CDATA #REQUIRED>
  <!ATTLIST ApplicationWindow height CDATA #REQUIRED>
  <!ATTLIST GuiElement x CDATA #REQUIRED>
  <!ATTLIST GuiElement y CDATA #REQUIRED>
  <!ATTLIST GuiElement width CDATA #REQUIRED>
  <!ATTLIST GuiElement height CDATA #REQUIRED>
  <!ATTLIST Button enable CDATA "yes">
  <!ATTLIST Button text CDATA #REQUIRED>
  <!ATTLIST TextInput enable CDATA "yes">
  <!ATTLIST TextInput passwordType CDATA "no">
  <!ATTLIST TextInput text CDATA #REQUIRED>
  <!ATTLIST Checkbox enable CDATA "yes">
  <!ATTLIST Checkbox checked CDATA "no">
```

```
  <!ATTLIST Label text CDATA #REQUIRED>
]>
```

5.7 Neimplementovaná funkčnost

Během implementace se vyskytly dva problémy, kvůli kterým jsem nebyl schopen implementovat některé funkčnosti knihovny GuiTestLib. V těchto případech je vyhozena výjimka s informací o tom, co musí být implementováno. Nejzávažnější problém se objevil při ovládání kurzoru myši po obnovení virtuálního stroje ze snapshotu. VNC klient se dokázal připojit k počítači, mohl ovládat jeho klávesnici, ale nedokázal manipulovat s kurzorem myši. Z tohoto důvodu nemohlo proběhnout rozpoznávání GUI objektů z windows, protože se aplikace dostávala při prozkoumávání oblasti do jiných stavů. Problém je vyřešen tak, že se každé window vyhodnotí jako objekt typu `UnknownObject`, který bude obsahovat informace pouze o svých rozměrech.

Druhým problémem je implementace metody `ExecuteCommand` pro vykonání zaslaného příkazu na virtuálním stroji. Z tohoto důvodu je do knihovny GuiTestLib přidána metoda `StartApplication` s ukazatelem na funkci, která spustí aplikaci. Uvnitř funkce se může využít metody knihovny pro kliknutí na určitou pozici nebo se lze připojit přes nějakého VNC klienta a testovanou aplikaci spustit manuálně.

Kapitola 6

Ověření funkčnosti implementované knihovny

V této kapitole je vyhodnocené testování funkčnosti knihovny pro vyhledávání windows. Testování proběhlo na virtuálním stroji s operačním systémem *Ubuntu 9.10*. Detekované oblasti byly pro znázornění zvýrazněny pomocí knihovny OpenCV. Mezi testované aplikace patří *Calculator*, *CD/DVD Creator*, *Display Preferences* a *Gedit*. Výsledky testování jsou zobrazeny na obrázcích v příloze B. V příloze C je zdrojový kód, který byl použitý na testování všech aplikací. Pro spuštění je nutné mít nainstalované nástroje Libvirt, LibVNCClient, OpenCV a Tesseract. Nevýhodou knihovny GuiTestLib je její časová náročnost. Detekování oblastí u testovaných aplikací trvalo 30-60 minut. Rozpoznání windows proběhlo úspěšně, až na dva malé nedostatky. U aplikace Gedit byla falešně detekována oblast, ve které se nenachází žádný element. Tento nedostatek se ale odstraní během klasifikace window. Druhou chybou je u aplikace CD/DVD Creator sloučení dvou oblastí (cdrom0 a floppy0) do jedné.

Kapitola 7

Závěr

Cílem bakalářské práce bylo vytvořit knihovnu pro automatizované testování GUI. Postupně byly uvedeny všechny důležité technologie a principy, s kterými se lze setkat při testování GUI aplikací. Byly rozebrány možnosti jakými lze docílit virtualizace, komunikace s virtualizovaným strojem, rozpoznávání a detekce obrazu. Původně jsem měl o výsledku této práce jiné představy. Kvůli problému s obnovováním snapshotů jsem většinu svého úsilí směřoval na co nejpřesnější detekci oblastí, ve kterých se nachází nějaký GUI element. Při vyhledávání a detekování je nevýhodou, že nemáme žádný soubor pravidel, který by definoval způsob chování různých GUI objektů, jejich maximální a minimální velikosti a vlastnosti, které musí splňovat. Za tímto cílem byly navrženy tzv. *UI Guidelines*[14]. Většina GUI aplikací ale nedodržuje tento standard, proto ho při implementaci knihovny nebylo využíváno.

7.1 Návrhy na rozšíření

Do budoucna by bylo zajímavé dokončit implementaci funkce pro rozpoznávání GUI objektů. Bylo by nutné buď vyřešit problém se snapshoty, nebo rozpoznávat objekty pouze na základě zpracování obrazu. Pokud by se tak stalo, bylo by dobré doplnit některé běžně vyskytující se objekty GUI, které lze knihovnou detekovat. Při vytváření nového typu objektu by bylo nezbytné implementovat jeho rozpoznání z window. Nový element by měl obsahovat své typické vlastnosti a metody, které by byly k dispozici pro jeho ovládání.

Metody pro detekování windows by se mohly rozšířit o další způsoby. Rozpoznávání souvisejícího textu by se mohlo detekovat přes označování textu myší, při které označený text změní svojí barvu. Při detekování oblastí jako jsou tlačítka nebo textové pole by se dalo využívat změny kurzoru myši (například u tlačítek se kurzor myši většinou změní na ikonu ruky). Přístup by vyžadoval přesnou detekci s minimálním počtem chyb.

Literatura

[1] Introduction to Penetration Testing [online]. https://community.rapid7.com/docs/DOC-2248, 2013-04-17 [cit. 2014-05-11].

[2] Unit, Integration, and Functional Testing [online]. http://docs.pylonsproject.org/projects/pyramid/en/1.3-branch/narr/testing.html, 2014-12-05 [cit. 2014-05-14].

[3] Internal drivers [online]. http://libvirt.org/drivers.html, [cit. 2014-04-25].

[4] Ammann, P.; Offutt, J.: *Introduction to software testing.* Cambridge University Press, 2008, iSBN 13 978-0-521-88038-1.

[5] Bradski, G.; Kaehler, A.: *Learning OpenCV: Computer Vision with the OpenCV Library.* O' Reilly Media, Inc., 2008, iSBN 978-0-596-51613-0.

[6] Brooks, P. A.; Memon, A. M.: *Automated GUI Testing Guided By Usage Profiles.* In Proc. of the Twenty-Second IEEE/ACM Intal Conference on Automated Software Engineering (Atlanta, Georgia, USA), 2007, iSBN 978-1-59593-882-4.

[7] Burnstein, I.: *Practical software testing: a process-oriented approach.* Springer-Verlag New York, Inc., 2003, iSBN 0-387-95131-8.

[8] Engström, E.; Runeson, P.: *A Qualitative Survey of Regression Testing Practices.* Department of Computer Science, Lund University, 2010, iSBN 978-3-642-13791-4.

[9] Hlaváček, M.: *Výkonnost síťové komunikace ve virtualizovaných prostředích.* bakalářská práce, FIT VUT v Brně, 2010.

[10] Kaner, C.; Falk, J.; Nguyen, H. Q.: *Testing Computer Software, 2nd Edition.* John Wiley & Sons, 1999, iSBN 0471358460.

[11] Memon, A. M.; Soffa, M. L.; Polack, M. E.: *Coverage Criteria for GUI Testing.* Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, 2001, iSBN 1-58113-390-1.

[12] Myers, G. J.; Sandler, C.; Badgett, T.; aj.: *The Art of Software Testing, Second Edition.* John Wiley & Sons, 2004, iSBN 0-471-46912-2.

[13] Richardson, T.; Stafford-Fraser, Q.; Wood, K. R.; aj.: *Virtual Network Computing.* IEEE Internet Computing Volume 2, 1998, iSBN 0-8186-7800-3.

[14] Smrčka, A.: Testování aplikací využívající síť, testování grafického uživatelského rozhraní. Fakulta informačních technologií, Vysoké učení technické v Brně, 2014-04-30.

[15] Virzi, R. A.: *Refining the test phase of usability evaluation: how many subjects is enough?* In Proceedings of Human Factors & Ergonomics Society, 1992.

Příloha A

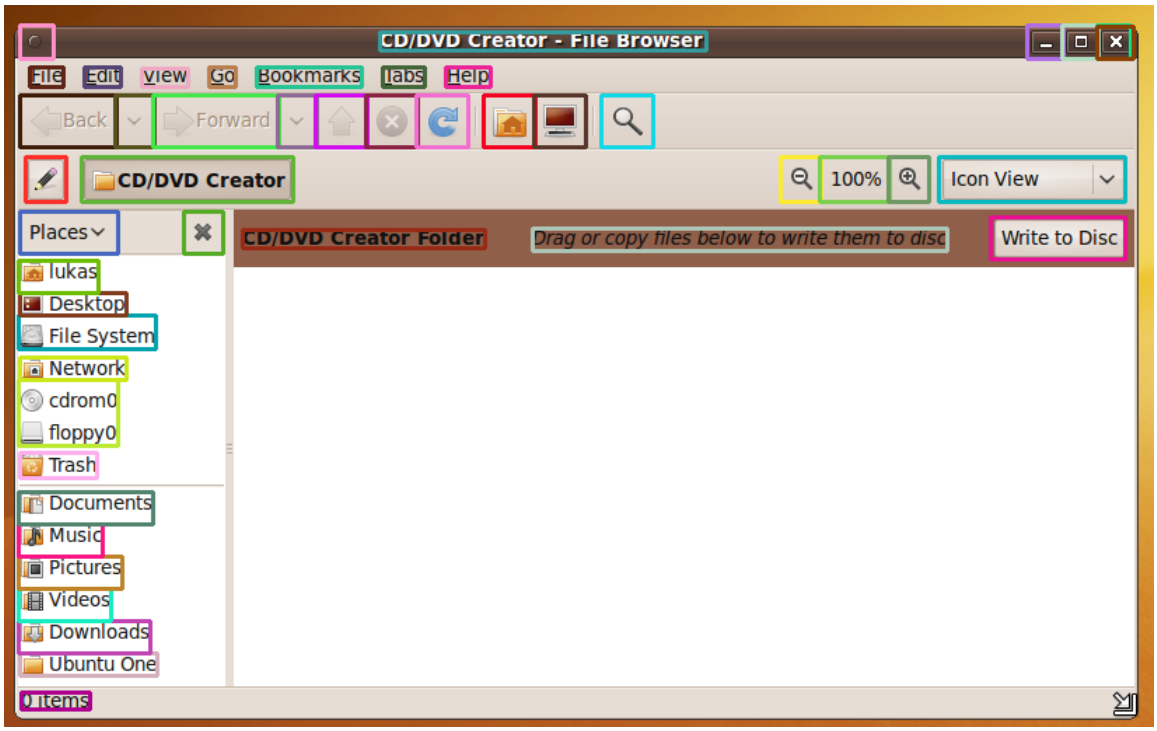
Obsah CD

Obsahem přiloženého cd jsou adresáře s následujícím obsahem.

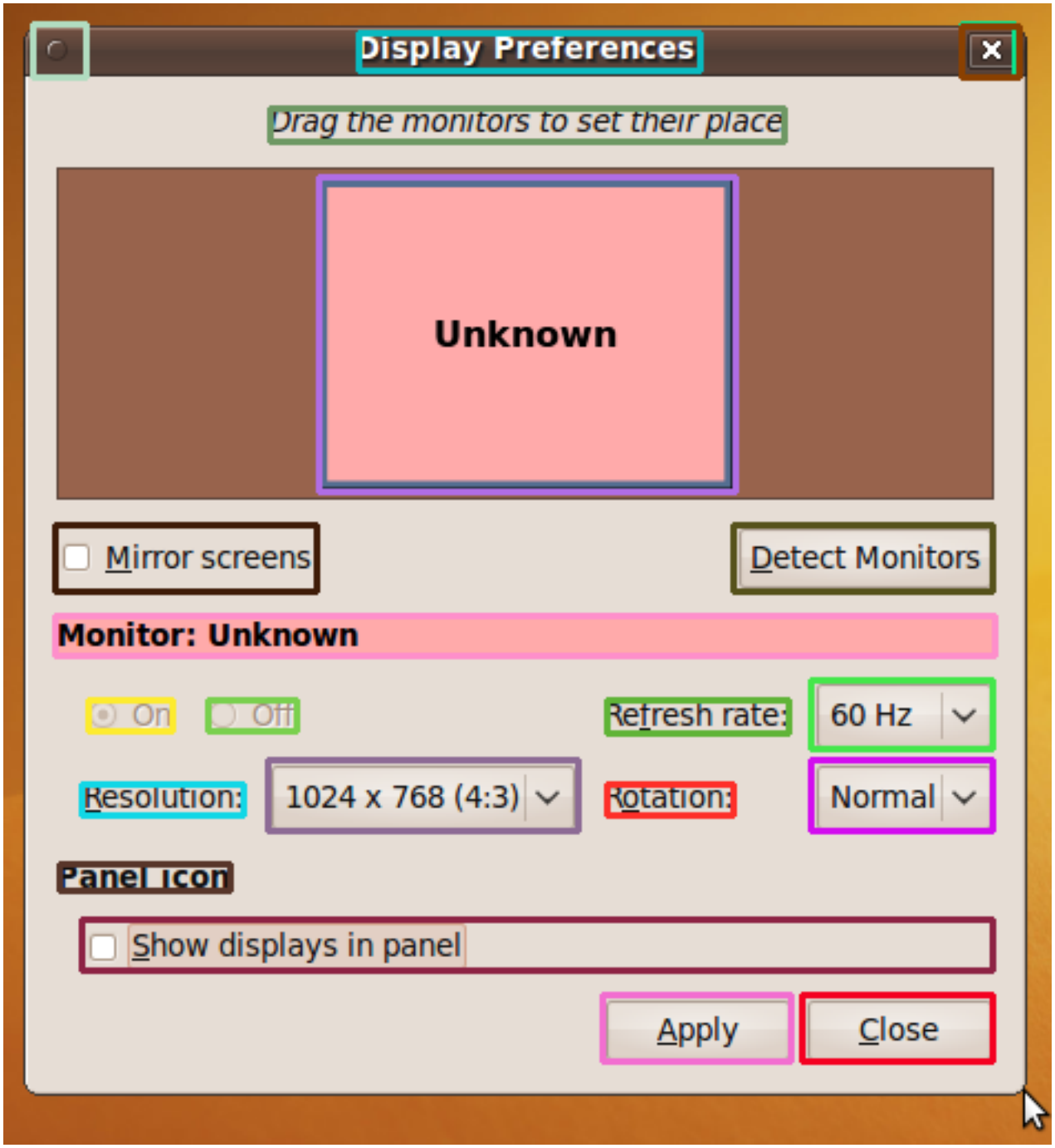
- **src** - zdrojové soubory knihovny GuiTestLib
- **include** - hlavičkové soubory knihovny GuiTestLib
- **example** - příklad použití knihovny GuiTestLib
- **tex** - soubory se zdrojovým kódem pro LaTeX obsahující text této práce

Příloha B

Obrázky detekovaných oblastí při testování

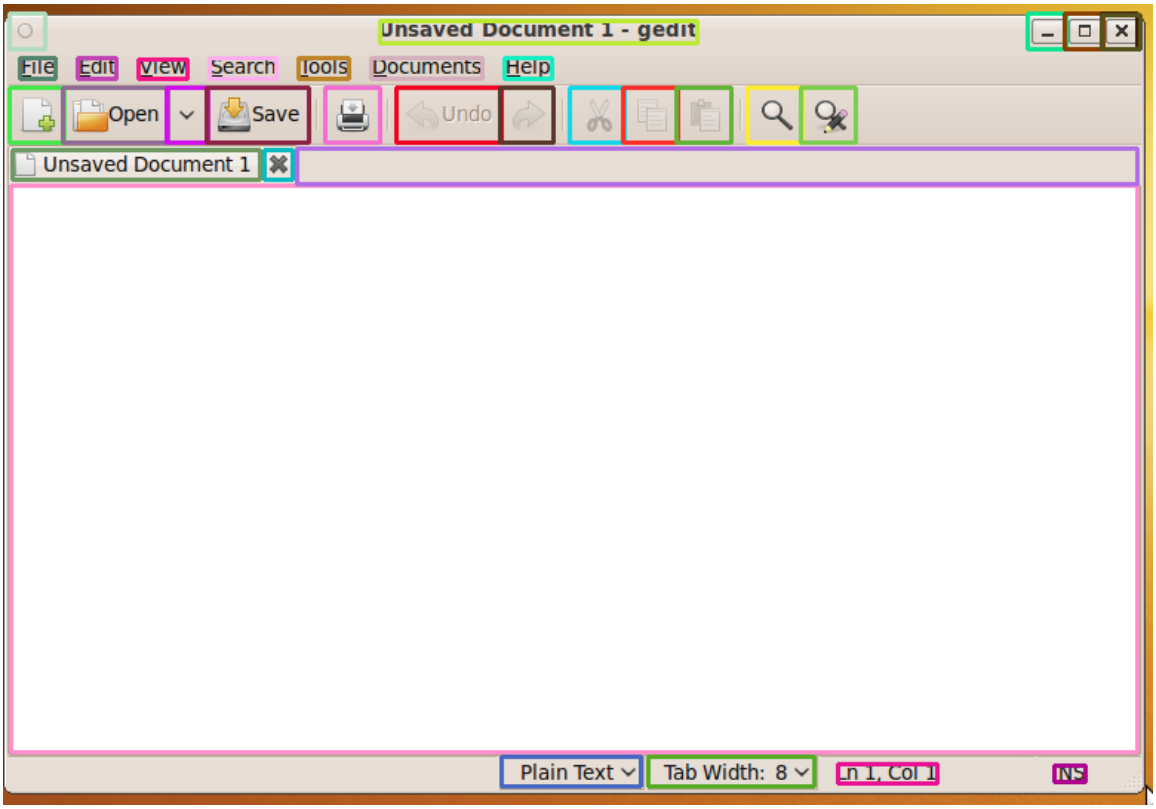


Obrázek B.1: Detekované oblasti aplikace CD/DVD Creator



Obrázek B.2: Detekované oblasti aplikace Display Preferences

Obrázek B.3: Detekované oblasti aplikace Calculator



Obrázek B.4: Detekované oblasti aplikace Gedit

Příloha C

Kód pro otestování detekovaných oblastí

```
#include <iostream>
#include "guitestlib.h"
const string xmlconfig =
  "<domain type='kvm'>"
  "  <name>UbuntuVM</name>"
  "    <uuid>c7a5fdbd-cdaf-9455-926a-d65c16db1810</uuid>"
  "  <memory unit='KiB'>524288</memory>"
  "  <currentMemory unit='KiB'>524288</currentMemory>"
  "  <vcpu>2</vcpu>"
  "  <os>"
  "    <type arch='x86_64' machine='pc'>hvm</type>"
  "    <boot dev='hd'/>"
  "  </os>"
  "  <devices>"
  "    <input type='tablet' bus='usb'/>"
  "    <emulator>/usr/bin/qemu-system-x86_64</emulator>"
  "    <disk type='file' device='disk'>"
  "      <source file='/home/lukas/Plocha/ubuntu.img'/>"
  "      <target dev='hda'/>"
  "    </disk>"
  "    <interface type='network'>"
  "      <source network='default'/>"
  "    </interface>"
  "    <graphics type='vnc' port='5901'>"
  "      <listen type='address' address='127.0.0.1'/>"
  "    </graphics>"
  "    <channel type='unix'>"
  "  <source mode='bind' path='/var/lib/libvirt/qemu/f16x86_64.agent'/>"
  "  <target type='virtio' name='org.qemu.guest_agent.0'/>"
  "    </channel>"
  "  </devices>"
```

```
  "</domain>";

void StartAppMethod(){
  cout << "Start application for testing and press key\n";
  getchar();
  return;
}

int main(int argc, char **argv) {
  try {
    GuiTestLib *g = new GuiTestLib();
    LibvirtHypervisor *vm = new LibvirtHypervisor(xmlconfig.c_str());
    MethodResult *r = new MethodResult();
    r = g->StartVMAndWaitForLoading(vm, 30);
    if(!r->result) {
      cerr << r->errorMessage;
      return -1;
    }
    LibVNCClient *vncClient = new LibVNCClient();
    r = g->ConnectVNCClientToRunningVM(vncClient);
    if(!r->result){
      cerr << r->errorMessage;
      return -1;
    }

    g->StartApplication(StartAppMethod);
    g->RecognizeObjects();
    g->ShowObjects();
    g->SaveActualSceneToXML("Data/XML/scene.xml");
    r = g->DestroyVM();
    if(!r->result){
      cerr << r->errorMessage;
      return -1;
    }
  }
  catch(TestingGUIException &e) {
    cerr << "TestingGuiException catched with description: " << e.what();
    return -1;
  }
}
```
